Pantheon: Preemptible Multi-DNN Inference on Mobile Edge GPUs

Lixiang Han
Department of Computer Science
City University of Hong Kong
Hong Kong SAR, China

Zimu Zhou
School of Data Science
City University of Hong Kong
Hong Kong SAR, China

Zhenjiang Li
Department of Computer Science
City University of Hong Kong
Hong Kong SAR, China

ABSTRACT

GPUs are increasingly utilized for running DNN tasks on emerging mobile edge devices. Beyond accelerating single task inference, their value is also particularly apparent in efficiently executing multiple DNN tasks, which often have strict latency requirements in applications. Preemption is the main technology to ensure multitasking timeliness, but mobile edges primarily offer two priorities for task queues, and existing methods thus achieve only coarse-grained preemption by categorizing DNNs into real-time and best-effort, permitting a real-time task to preempt best-effort ones. However, the efficacy diminishes significantly when other real-time tasks run concurrently, but this is already common in mobile edge applications. Due to different hardware characteristics, solutions from other platforms are unsuitable. For instance, GPUs on traditional mobile devices primarily assist CPU processing and lack special preemption support, mainly following FIFO in GPU scheduling. Clouds handle concurrent task execution, but focus on allocating one or more GPUs per complex model, whereas on mobile edges, DNNs mainly vie for one GPU. This paper introduces Pantheon, designed to offer fine-grained preemption, enabling real-time tasks to preempt each other and best-effort tasks. Our key observation is that the two-tier GPU stream priorities, while underexplored, are sufficient. Efficient preemption can be realized through software design by innovative scheduling and novel exploitation of the nested redundancy principle for DNN models. Evaluation on a diverse set of DNNs shows substantial improvements in deadline miss rate and accuracy of Pantheon over state-of-the-art methods.

CCS CONCEPTS

• **Computer systems organization** → **Real-time system architecture**; • **Computing methodologies** → **Neural networks**.

KEYWORDS

Mobile Edge Systems, GPU Scheduling, Preemption, Deep Learning

ACM Reference Format:

Lixiang Han, Zimu Zhou, and Zhenjiang Li. 2024. Pantheon: Preemptible Multi-DNN Inference on Mobile Edge GPUs. In *The 22nd ACM Conference on Mobile Systems, Applications, and Services (MobiSys '24), June 3–7, 2024, Tokyo, Japan.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

MobiSys '24, June 3–7, 2024, Tokyo, Japan

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Mobile edge (mEdge) devices, such as the NVIDIA Jetson series and other systems-on-chips, are increasingly vital for running deep neural network (DNN) tasks in various autonomous machines and applications [31, 44, 53]. The GPU has become the primary executor and accelerator for running DNNs on these devices [50]. Beyond accelerating the inference of a single task, the significance of GPU also begins to be reflected more in its ability to efficiently run multiple DNN tasks in recent applications [39, 68, 69] like autonomous driving, roadside units, virtual reality, etc., where many DNN tasks often have strict latency requirements [6, 69].

Preemption [65] is a main technique to ensure the timeliness of running multiple concurrent tasks, by allowing urgent tasks to interrupt ongoing ones. This technology ensures that a processor can fully exploit its inherent computing capability, since its hardware spends huge investment ($41.82 billion on GPUs per year [26]) to manufacture and improve, but the achieved task timeliness can be easily compromised due to inefficient scheduling, thereby wasting computing power. Violating latency requirements of DNN tasks can further lead to system failures (*e.g.*, incorrect system decisions due to outdated results [39]) and even safety issues (*e.g.*, danger alerts in autonomous driving or roadside units [13]). Therefore, effective preemption is also crucial to enhance the reliability of mobile edge systems and applications.

However, existing mobile edge GPUs lack efficient preemption support [16, 66], and commercial deep learning frameworks (such as PyTorch [54] and TensorFlow [1]) usually provide only two priorities (high and low) for task processing queues (GPU streams) [2]. This results in recent methods achieving merely simple, two-tier preemption [16] by classifying DNN tasks into two types: real-time (high-priority) and best-effort (low-priority), and allowing a real-time task to preempt best-effort tasks [16]. However, the efficacy drops significantly when other real-time tasks run concurrently (as additional high-priority contenders), but this is already common in applications, such as the concurrent real-time DNN tasks for object detection, lane detection, road segmentation and depth estimation on autonomous vehicles [39], simultaneous sensing [67] and recognition [63] in augmented reality [69], concurrent computer vision and emotion recognition in robots [9], and more.

Can we apply solutions developed for other platforms to mobile edge GPUs? Traditional mobile devices primarily rely on CPUs [38]. Even for DNN tasks, GPUs mainly serve as supportive processors, providing at most comparable performance to CPUs [62]. These

devices typically lack specialized preemption support and adhere to a first-in-first-out (FIFO) policy for GPU scheduling, though they can expedite DNN inference through cross-processor scheduling [29, 30, 69]. In contrast, mobile edge GPUs are explicitly designed for deep learning, capable of managing DNN tasks independently. High-performance platforms such as clouds or data centers do manage concurrent tasks, often with multiple GPUs dedicated to complex DNN tasks and focusing on allocating one or more GPUs per model [37]. However, the scenario is distinct for mEdge devices, where multiple DNNs compete for a single powerful GPU. This unique setting requires fine-grained and timely preemption to efficiently manage these competing tasks.

Hence, mEdge GPUs need their own preemption support, which would allow real-time DNN tasks to preempt not only best-effort tasks (considered the lowest priority), but also each other, depending on their respective priorities (such as the remaining time to their deadlines [43]). To achieve this, we encounter two challenges:

1) Limited backend support. The existing two-tier preemption seems to be caused by insufficient task queue (GPU stream) priorities. However, current mobile edge GPU backends, including their hardware and corresponding software processing in drivers and deep learning frameworks, typically adopt a one-DNN-per-stream execution strategy [64, 66]. That is, a DNN task, once dispatched into a GPU stream, enters a standard GPU routine, and the GPU stream priority is tied to this task. However, the priority of real-time tasks may change due to the arrival of other new real-time tasks, while the priority of the tied stream is not reconfigurable (§2). Hence, simply increasing the number of GPU stream priorities cannot directly resolve the issue. A new suitable preemption logic, compatible with the limited preemptive support from the underlying GPU backend, is needed.

2) DNN-oriented context switch. Even if we had such a preemption solution, frequent pausing and resuming of DNN inference due to preemption would further introduce two unique issues. First, the context switching overhead, including checkpoint saving and resumption, can nullify the benefits of preemption if the overhead is high. Second, a preempted DNN task may not complete the remaining part before the deadline upon resumption. Any real-time task output that misses the deadline becomes invalid [66]. Therefore, to effectively support preemption, DNN should further have flexible online model adjustment capabilities after context switching back to ensure its timeliness.

In this paper, we address these challenges by designing a system, called Pantheon, that works with commodity mobile edge GPUs to provide a general and fine-grained preemption for running concurrent DNN tasks. We discover that the two-tier stream priority is already sufficient, which can still be used for real-time tasks to preempt best-effort ones. Our main observation is that we can enable preemption logic for real-time tasks on top of high-priority streams via novel scheduling in software. In this way, when the tasks that need to be executed are determined, high-priority streams only serve as a pipeline to convey them to the GPU for execution. Meanwhile, we also leverage the structured nature of DNN inference to slice DNN models into appropriate chunks so that Pantheon can preempt tasks at the chunk level, thereby achieving timely (microsecond level) preemption scheduling.

In addition to preemption logic, another unique challenge in designing preemptive DNN inference is that if the DNN is preempted, its remaining part may not be completed before the deadline after resuming. The redundancy in DNNs allows for skipping certain operations with minimal impact on model accuracy [45, 59]. However, previous studies [42] exploring compressed model variants (with varying complexity and accuracy) for fixed-deadline inference are not suitable for us, because then we would need to run a new variant from scratch each time. In this way, even if we run a very small variant, it may still fail to complete when deadlines are tight or model accuracy is low. Therefore, for efficient preemption, each model variant should be able to utilize the intermediate results generated by its larger counterpart and continue the inference process seamlessly, which is called *nested redundancy*. In Pantheon, we exploit nest redundancy under a new optimization framework with early exits [24, 34]. This enables the DNN to reuse the results prior to preemption and adjust the remaining sliced model chunks to meet deadline while maintaining good accuracy.

Experiment. We develop a prototype of Pantheon[1] and evaluate its performance on NVIDIA Jetson NX and Nano across a broad range of applications. These include nine DNN tasks for smart traffic, service robot and UAV ground station three applications. Experiments show that Pantheon outperforms the classical task scheduling methods in real-time system areas [4, 20, 36] and the state-of-the-art DNN-oriented scheduling RTm-DL [42] on mEdge devices, improving task deadline miss rates by up to 99.4% and 97.5%, respectively. Meanwhile, Pantheon can enhance the accuracy performance by 315.5% and 46.0%, respectively. We also deploy Pantheon on an autonomous car for field evaluation, achieving consistently good performance, with a 3.59% deadline miss rate and 95.96% accuracy. Finally, Pantheon is lightweight, with each preemption taking about 46.2–218.9 μs to complete. In summary, this paper makes the following contributions:

- We propose a novel and general preemption design for multi-DNN inferences on mobile edge GPUs, accommodating the growing trend of more concurrent DNNs in emerging mobile edge applications.
- We address unique challenges in Pantheon. Our design can work on top of the deep learning framework without modifying the framework and GPU driver, making it easy (as a plug-in) to upgrade massive mEdge devices to possess preemption capabilities.
- We develop a prototype system and extensive evaluations show significant performance gains compared to the latest GPU task scheduling in real-time systems and the state-of-the-art mobile edge design.

2 BACKGROUND AND MOTIVATION

The commodity mEdge GPU backend, including its hardware and corresponding software processing in the driver and deep learning framework, offers limited support for preemptive scheduling of multi-DNN tasks, as explained below.

1) Limited preemption support. The main goal of existing DNN scheduling on GPU is to strategically distribute computation among

[1]Pantheon is open-sourced at https://pantheoninfer.github.io.

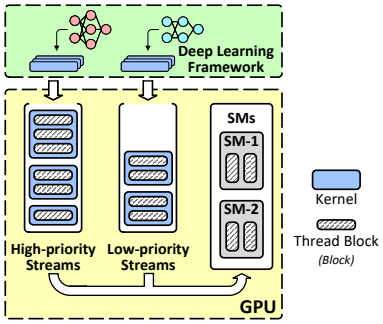

Figure 1: DNN processing in mobile edge GPUs.

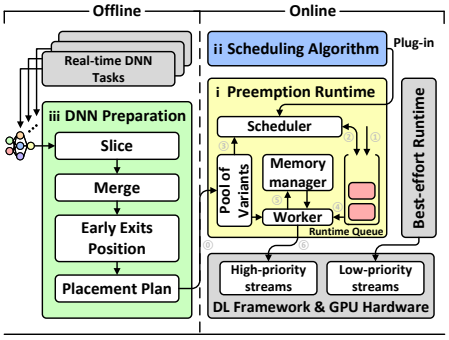

Figure 2: Overview of the Pantheon design.

multiple GPU computing units. It promotes parallel processing and aims to maximize throughput.

Single DNN. We start with the single DNN case. For GPU execution, each DNN is decomposed into *kernels* [2]. These kernels will be executed in parallel by multiple *GPU threads*, which are further organized into *thread blocks* and assigned to computing units, stream multiprocessors (SMs), to run.

Multi-DNNs. By default, multiple DNNs are handled separately [64, 66]. Each DNN is pushed into a *GPU stream*, *i.e.*, a logical task queue in GPU programming. A GPU usually provides dozens of GPU streams. All the kernels pushed into the same GPU stream are executed following a FIFO policy. To provide flexibility in scheduling tasks across different GPU streams, the existing mobile edge GPU backend provides a simple two-tier preemption mechanism (see Figure 1), *i.e.*, lower-priority streams can dispatch blocks to SMs only when higher-priority streams are empty [2].[2] Therefore, in recent methods, real-time tasks (in high-priority streams) can only preempt best-effort tasks (in low-priority streams).

In short, DNN execution on mobile edge GPUs opts for throughput, with fine-grained (thread-block-wise) *intra-stream* scheduling to fully utilize SMs but coarse-grained *inter-stream* scheduling. Given such a backend design, problems faced with fine-grained scheduling across GPU streams appear to stem from a lack of adequate GPU stream priorities, but we find that directly providing more levels of stream priority does not fundamentally solve the problem, for the following reason. Because for real-time DNN tasks, their priorities may change over time due to other newly incoming real-time tasks. However, once a DNN task is dispatched to a stream, the task's priority is tied to the GPU stream to which it is assigned. Stream priority is not reconfigurable [49], which would otherwise greatly increase hardware and driver complexity, preventing us from adjusting the priority of real-time tasks.

Targeted preemption design. Hence, we need a new preemption solution, compatible with the limited preemption support from the underlying backend. To this end, we lean to a software solution on top of the mobile edge GPU backend for fine-grained preemption. It should allow real-time tasks to preempt each other based on their priorities. Real-time tasks can be periodical or non-periodical. Fine-grained preemption is important for both, as the arrival of periodical tasks may not adhere to fixed intervals due to unavoidable uncertainty or jitter in the input preprocessing [42], which makes the static scheduling unable to stably guarantee their timeliness. This uncertainty can be more severe for non-periodical tasks. On the other hand, the new preemption solution should also be backward compatible with existing preemption enabling real-time tasks to preempt best-effort tasks. We find that two stream priorities are actually sufficient, and we will detail our design of Pantheon to achieve this goal in §3.

2) Insights from CPU preemption. Even if we had such a preemption solution, the performance will suffer if its efficiency is low since it processes DNN tasks. In light of this, we draw inspiration from the preemption on CPUs. Preemption is mature on CPUs, which divides tasks into small *chunks* in time (time slicing) so that the CPU can switch between different tasks for multitasking [58]. Since CPUs are opted for flexibility, the underlying hardware and operating system support lightweight context switching, *i.e.*, CPU preemption only requires saving the task context from dozens of registers (*e.g.*, 32 registers on ARM64 [3]) to memory.

Following the same principle, we can also slice the DNN into chunks, hoping for fine-grained cross-DNN preemption in time. A chunk is different from a thread block and is typically larger than it. However, GPU involves massively parallel processing, which produces massive intermediate results, *e.g.*, more than ten thousands GPU registers [51]. Thus, DNNs cannot be sliced into chunks arbitrarily. Otherwise, excessive intermediate results will be saved and restored, which will notably slow down context switching and thus increase the latency of DNN inference.[3] Pantheon will carefully address this issue, which adopts the principle of CPU preemption and caters for the unique DNN characteristics.

3 SYSTEM DESIGN

Figure 2 shows the Pantheon design that works between the deep learning framework and the applications.

1) Online module. This module is the core component for enabling preemptive DNN inference. It includes our proposed (i) preemptive runtime framework, which consists of a dedicated worker, memory manager, runtime queue, etc., as well as (ii) core preemptive task

[2] In addition, the execution of blocks in the same GPU stream always follows FIFO, and the running of blocks in different GPU streams but with the same priority can be unordered, which is scheduled by the GPU driver to maximize GPU throughput.

[3] Taking GoogLeNet as an example, the intermediate results between chunks are 54.1 MB on average if sliced uniformly (see Figure 6), which cause a delay of about 0.88 ms when moving intermediate results from the cache to the memory during context switches on a Jetson device. Since the deadline of real-time tasks in mEdge applications is usually at the millisecond level, this delay cannot ignored and should be reduced, *e.g.*, it can be reduced to approximately 0.37 ms based on our slicing design in §3.2.1.

scheduling algorithm (§3.1). The internal workflow of the runtime framework (⓪–⑥) will be detailed in Figure 3. The runtime connects high-priority GPU streams to enable preemption between different real-time tasks. Consistent with existing work, low-priority streams are reserved for best-effort tasks. This online module can be installed as an intermediary layer between applications and the GPU backend without requiring modifications to its deep learning framework and the underlying GPU hardware and driver.

2) Offline module. This module contains two necessary (iii) preprocessing of the DNN to support preemption (§3.2), which is a one-time effort when the DNN is registered in the system.

The application may have its internal controls to obtain better quality of service (QoS), such as skipping certain non-keyframe processing. Since Pantheon aims to provide a general-purpose preemptive DNN inference service for upper-layer applications, we design it as a middleware between the application and the device backend. If an application requires QoS control, it will be performed at the application layer and this decision will not be deferred to Pantheon. Therefore, we consider all the input tasks to Pantheon should be performed. Below we introduce the detailed design of each module and use the following priority setting in Pantheon:

- **Real-time tasks**: tasks with shorter remaining time to their deadline will have higher priority, which are allowed to preempt each other with Pantheon.
- **Best-effort tasks**: they (*e.g.*, in-car entertainment or assistant services [8]) all have lower priority than real-time tasks. Similar to existing work [16], they are handled in low-priority GPU streams following FIFO.

3.1 Online Preemptive Design

We first describe the runtime architecture to enable the preemption of DNN tasks in §3.1.1 and then introduce the core preemptive task scheduling algorithm in §3.1.2.

3.1.1 Preemption runtime. Figure 3 illustrates the runtime preemption architecture proposed in Pantheon.

1) Principle and design. Once the runtime dispatches a DNN to GPU streams, it enters the standard GPU inter-stream scheduling routine and we lose further control. Therefore, we should complete the entire preemption logic in the runtime.

Our main idea of achieving preemption is to timely *refresh* (*sort*) all DNN chunks in the runtime queue in order of priority from high to low, where all chunks of a DNN have the priority of that DNN task. Then the next DNN chunk to run will always be from the task with the highest priority. Specially, similar to time slicing in CPU preemption [58], we also slice each DNN into chunks. Meanwhile, we add early exits in each DNN to generate multiple model variants in the registration stage of the model and ⓪ store them in a pool of variants (see Figure 3). How to achieve good DNN slicing and variant generation for preemption will be introduced in §3.2. With Pantheon, DNNs to be executed first ① enter the runtime queue. The scheduler then ② refreshes the runtime queue by sorting the tasks and ③ selecting the suitable variants (based on the algorithm in §3.1.2). Once the GPU has free computing resources, the worker ④ fetches the next chunk in the queue and ⑤ collaborates with the memory manager to ⑥ dispatch the chunk to the device backend for

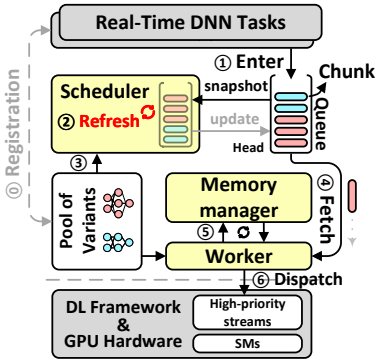

Figure 3: The preemption runtime in Pantheon.

execution. If the new DNN has the highest priority, it will be placed at the queue head, which is equivalent to interrupting the DNN task that was previously at the head, thereby achieving preemption. The chunk-wise dispatching ensures that a lower-priority DNN task can be interrupted in the middle of its execution.

In Figure 3, after the first chunk of the (pink) DNN is dispatched for execution, another (skyblue) DNN enters the queue of a higher priority. After scheduling, its chunks will be placed at the queue head to run first, which interrupts the execution of the (pink) DNN task. Overall, the runtime works as follows. All real-time DNN tasks first enter the runtime queue. The worker always dispatches the first chunk of the DNN in the head of the queue for execution when GPU has available computing resources, and the memory manager saves the state of preempted tasks, *i.e.*, the generated intermediate inference results from the executed chunks. Whenever a new DNN task enters the queue, the scheduler is triggered to refresh the queue based on the priorities of tasks. However, to ensure the correct operation of the runtime, we need to further address a task order inconsistency issue.

2) Task order inconsistency issue. When a new DNN enters the runtime, it is pushed to the end of the queue, and the scheduler is triggered to update the order of existing tasks. To this end, the scheduler takes a **snapshot** of the current task order and computes a new order. However, before the scheduling is completed, it is possible that the ongoing chunk in GPU is finished and the next chunk (denoted as u) of the current head task needs to be dispatched. However, since chunk u is in the snapshot, after the scheduler returns the new order, chunk u is still included. Because this chunk has been dispatched, its state data will be released after execution to conserve memory, and an attempt to run it again will result in a system error. We address this issue from two aspects.

- First, the scheduler needs to update the task order very quickly to minimize the chance that such chunks (*e.g.*, chunk u) occur, which will be introduced below.
- Second, we further introduce a vector $<\ldots, f^j_{n_i}, \ldots>$ in the memory manager, where $f^j_{n_i}$ a pointer pointing to the first undispatched chunk j of each existing DNN n_i, based on which the worker can also skip the chunks dispatched already as a remedy for this problem.

The above two designs are indispensable. If we only had the vector, the overall scheduling would be very slow, which could lead

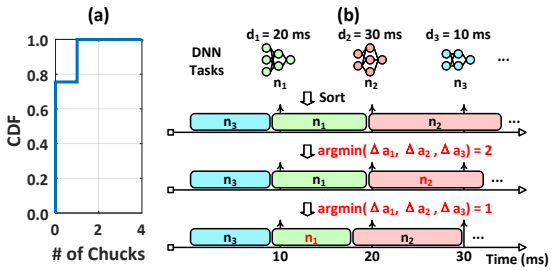

Figure 4: (a) CDF of task order inconsistency occurs. (b) Illustration of the preemptive scheduling execution.

to frequent device timeout errors. However, only adopting a faster algorithm does not completely avoid the task order inconsistency issue. Fortunately, due to the efficiency design, our scheduling algorithm (in §3.1.2) can be finished in the μs level, and the inconsistent task order does not occur frequently during execution. Throughout all our evaluation in §5, Figure 4(a) shows that in only 24.51% of the cases, at most one chunk suffers from the inconsistent order issue, which can be corrected by our remedy mechanism easily.

3.1.2 Preemptive scheduling algorithm. We then introduce the scheduling algorithm design used by the scheduler in the runtime. To determine the order to execute tasks with deadlines, the *earliest-deadline-first* policy has been extensively studied in the real-time system domain and proven that it can derive the optimal order of task execution when preemption is allowed and a feasible schedule exists [43]. However, for DNN tasks, their complexity is much heavier and the deadlines are quite tight [42]. Merely adjusting the execution order may not be feasible in practice, which could still result in a high deadline miss rate [42].

1) Observation and formulation. Although DNN tasks are complex, recent work has identified the potential to slim down DNN models by generating smaller variants that slightly affect accuracy but greatly reduce running latency [11, 59]. In Pantheon, we exploit this opportunity by jointly determining 1) the order of each task in the queue and 2) the appropriate variant with a smaller size for each DNN.

To this end, for each DNN n_i, we preprocess it to generate multiple (K) smaller variants $\{n_i^j\}_{j=0}^K$, where n_i^0 is the original DNN and the size of n_i^j decreases as j increases. Generating these variants is non-trivial due to a unique issue from preemption. How Pantheon generates them will be introduced in §3.2.2. We first assume that these variants are available.

Given all the variants $\{n_i^j\}_{j=0}^K$ of each DNN n_i, their accuracy $\{a(n_i^j)\}_{j=0}^K$ can be profiled in advanced [34]. We then define two decision variables $\mathcal{C}$ and $\mathcal{V}$, where $\mathcal{C}$ records the order of each DNN n_i to execute, and $\mathcal{V}$ specifies the variant n_i^j to select for each DNN n_i in $\mathcal{C}$, to achieve:

$$\max_{\{\mathcal{C},\mathcal{V}\}} \sum_{n_i \in \mathcal{C}} a(n_i^j); \quad s.t.\ (i)\ n_i^j \in \mathcal{V},\ (ii)\ r_i \le d_i, \qquad (1)$$

where the objective is to select the appropriate variant n_i^j for each DNN task n_i according to the first constraint (i) to maximize the overall accuracy (Since Pantheon is designed to provide a general preemptive service and perform all input tasks, we use this overall accuracy as the optimization objective), and the second constraint (ii) to ensure that the execution time r_i of the remaining part of each DNN task n_i is within the deadline d_i of DNN task n_i.

2) Large search-space issue. However, it is difficult to solve the optimization efficiently because a joint consideration of the execution order and the variant version largely increases the search space. Specially, the problem's time complexity in Eq. (1) is $O(N! \prod_{i=1}^{N} K_i)$, where N and K_i are the number of DNNs and the number of variants per DNN, respectively. Our experiment finds that directly solving Eq. (1) is very slow and will cause system timeout errors.

3) Solution. Our idea in solving this problem is to first assume that all tasks are feasible for scheduling and adopt the earliest-deadline-first strategy [43] to obtain a preliminary execution order. If there is indeed some task that will miss the deadline in this order, then we select this DNN and/or other DNNs preceding it to switch to a smaller variant so that it can catch up with the deadline.

Therefore, selecting suitable DNN variants essentially converts this problem into a feasible problem in traditional real-time systems, and the earliest-deadline-first strategy further guarantees the effectiveness of the execution order. This way we can still consider both decision variables in Eq. (1) (*i.e.*, the execution order and variant of DNN tasks), but effectively decouple their dependency to speed up the answer search, which trades some optimality for the latency performance to make the scheduling tractable and practical.

With this solution, DNN tasks are first sorted based on deadline from closet to furthest, and we then scan these sorted tasks starting from the head. For each DNN n_i of order c_i, we can compute its finishing time r_i below:

$$r_i = \begin{cases} t(l_i, c_i), & \text{when } i = 1, \\ r_{i-1} + t(l_i, r_i), & \text{otherwise,} \end{cases} \qquad (2)$$

where l_i is the index to the beginning of the remaining part of DNN n_i that is not executed yet due to preemption, and $t(l_i, c_i)$ is execution time of n_i starting from l_i. Our scheduling algorithm then proceeds as follows.

Step-1): For the first task (when $i = 1$), r_i is the time to execute its remaining part, while for other tasks, r_i also adds the execution time of all the tasks in front.

Step-2): During calculating each r_i in Step-1), if a DNN n_k cannot be finished before deadline, among all tasks before n_k and n_k itself (from c_1 to c_k), we select one task whose next smaller variant has the least accuracy drop, and use such variant for this DNN task, which can make n_k start earlier.

Step-3): After the above update, the scheduler computes Eq. (2) for n_k again. If its deadline can be met, the scheduler proceeds for the following tasks. If its deadline still cannot be met, the scheduler repeats Step-2) to replace one more task by its next smaller variant until the deadline is met. If all DNNs from c_1 to c_k already use their smallest variants but n_k still misses its deadline, it means that n_k is not schedulable anyway, and the scheduler skips it and follow Step-2) and Step-3) to process the next DNN task.[4]

Figure 4(b) shows an example. Tasks are first sorted by deadline. The scheduler then scans them from left to right, calculates the

[4]After a task is processed and can meet the deadline already, it may be adjusted again due to subsequent tasks. Since only smaller variants will be used, tasks that already meet the deadline will not become overdue again.

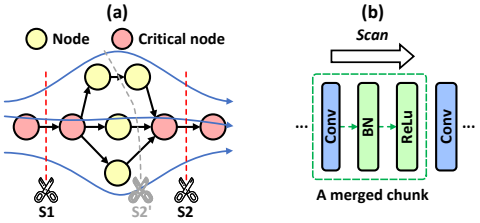

Figure 5: (a) Slice the DNN between critical nodes. (b) Merge computation-expensive and lightweight layers.

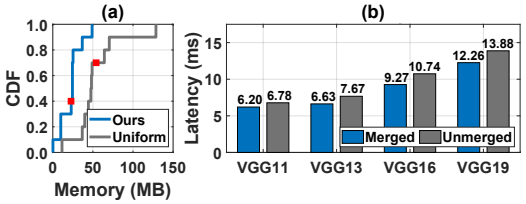

Figure 6: (a) Memory consumption using uniform and our slicing on branched DNNs *e.g.*, GoogLeNet. (b) Execution speedup for chain-structure DNNs by merging.

finishing time r_i of each task, and finds that n_2 is the first task to miss its deadline. Among n_3, n_1 and n_2, since n_2 has the least accuracy drop when changing it to its next smaller variant, the scheduler selects n_2 to change. After the changing, although n_2 can be completed earlier, it still exceeds the deadline. The scheduler therefore proceeds to select the next task whose next smaller variant causes the least accuracy drop, *e.g.*, n_3. By changing n_3 to the next smaller variant, n_2 can start earlier. The scheduler finds that n_2 can meet its deadline at this time, and its processing ends. The scheduler will then handle subsequent tasks similarly.

With this algorithm design, the search complexity can be reduced to $O(NlogN + \sum_{i=1}^{N} K_i)$. Our evaluation in §5.5 shows that each scheduling takes about 46–219 μs only.

3.2 Offline Preprocessing Design

The offline module of Pantheon contains two preprocessing operations, including 1) the DNN slicing to ensure timely chunk-wise interruption and minimize the overhead of context switching (§3.2.1), and 2) the generation of DNN variants required by our scheduling algorithm (§3.2.2).

3.2.1 DNN slicing for preemption. As aforementioned, if DNNs are not issued in chunks, once an entire DNN is dispatched into the GPU stream, it enters the standard GPU processing routine and our runtime loses the control to perform preemption further. Therefore, slicing DNN into chunks and performing chunk-wise interruptions ensure timely and fine-grained preemption. To slice DNNs for Pantheon, the primary consideration is to minimize the amount of intermediate data to store for the preempted chunks.

1) Design for slicing. DNN can be represented as a directed acyclic graph (DAG) [23]. Each DNN layer corresponds to a node in the graph, and the edges are constructed based on the connectivity between the layers. There are many paths from the start node (the first layer) to the end node (the last layer) in the DAG. Meanwhile, there exist many nodes through which all paths pass from the start node to the end node, and we define such nodes as *critical nodes*. As all paths are aggregated into the critical nodes, if the DNN is sliced at such nodes, the amount of intermediate data to store is small, *i.e.*, the intermediate data from just one node, such as S1 and S2 in Figure 5(a). Otherwise, the intermediate data from multiple nodes need to be stored, *e.g.*, S2' in Figure 5(a). This is helpful for most DNNs with branching structures. Therefore, we define the edges between two adjacent critical nodes as *slicing points* and use depth-first search to identify all slicing points. Then for each slicing point, we perform slicing to obtain a series of chunks for the DNN.

2) Enhancing execution efficiency. The slicing mechanism above considers only the storage overhead, but we find that it usually separates the *computation-expensive* layers (*e.g.*, convolution or fully-connected layers) with the *lightweight* layers (*e.g.*, batch-normalization or activation layers) into different chunks for chain-structured DNNs, *e.g.*, the VGG series. However, deep learning frameworks often optimize execution efficiency by fusing computation-expensive with lightweight layers [47], *e.g.*, PyTorch uses it for DAG optimization. Therefore, to exploit this optimization, we traverse all the chunks after DNN slicing. For any computation-expensive chunk *a*, the following chunks will be merged into *a* if they are lightweight chunks, as shown in Figure 5(b), which continues until the next computation-expensive chunk is encountered. We currently integrate this common fusion strategy into our slicing design, but it is not limited to this strategy. If more strategies are known in the future, we can integrate their fusion logic into our DAG optimization design and adjust accordingly.

The slicing and merging described above is a one-time effort that occurs when a pretrained DNN is registered to the system. Figure 6(a) shows that compared to the uniform division, our slicing can reduce the memory consumption per chunk by 60.7% on average for GoogLeNet. The layer merging mechanism can further improve the execution efficiency by 10.3% on average for various VGG models in Figure 6(b).

3.2.2 Generation of DNN variants. If a DNN is preempted, one unique challenge posed by preemption is that its remaining part may not be completed before the deadline after resuming. Existing work [33, 42, 46] suggests preparing multiple smaller variants of each DNN, *e.g.*, via compression [19, 28, 42]. But if so, we would need to run a new variant from scratch every time. In this way, even if we run a very small variant, it may still fail to complete when the deadline is tight or the model accuracy will be low. Thus, we need to consider how the intermediate result of the previous variant (before preemption) can be reused by the new variant without running the new variant from scratch, which is called nest redundancy (§1).

1) Observation. We find that the early exit mechanism [24, 34, 61] is well-suited for generating DNN variants when preemption is enabled, which inserts additional exits (*e.g.*, fully connected layers) into the DNN. Each exit can provide the same output as the original model exit, so we can choose to skip the rest of the model from any exit to end the model execution early. It is equivalent to running a smaller variant of this DNN, which reduces the execution latency.

With this mechanism, for each DNN task, we only need to maintain its original model with a set of early exits. By selecting different exits, we equivalently obtain different variants of this model. Thus, the intermediate results of the preempted chunk can be used to

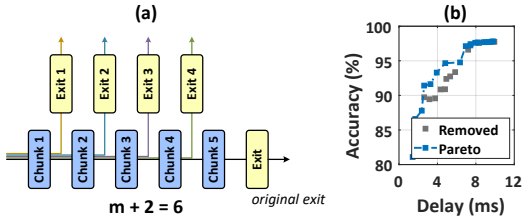

Figure 7: (a) Potential locations ($m = 4$) of adding early exits. (b) Pareto frontier for the accuracy and delay.

complete inference from any later exits after resuming execution by exploiting the nested redundancy. For example, in Figure 7(a), before preemption, the model aims to use the original exit, but its execution is preempted after the execution of "Chunk 2". After resuming, the intermediate results from "Chunk 2" can be further processed by "Chunk 3", even another variant is scheduled to use, *e.g.*, using "Exit 3 or 4". However, each exit also incurs additional memory overhead and achieves different accuracy. We should carefully plan where and how many exits to insert to achieve a good balance between overhead and accuracy.

2) Placement of early exits. If the DNN has $m+2$ chunks after slicing, in principle, we can insert an exit after each of the first m chunks, that is, m candidate locations to add early exits.[5] Therefore, we can initially add an early exit to each candidate location and then determine which exits should be preserved. The structure of each exit is set to be the same as the original exit of this DNN, but we can add one or two convolution and pooling layers to adjust their dimensions when added to the DNN. We then train these m exits by freezing the original DNN parameters and profile:

- the **accuracy** (a_j) and the **execution delay** (t_j),

when using each exit j after training. Then we define a decision variable $\boldsymbol{p} = (p_1, \cdots, p_{m+1})$, where each $p_j \in \{0, 1\}$ indicates whether the j-th exit should be preserved. The subscript up to $m+1$ means the original DNN exit is allowed to remove. We consider the following aspects to optimize $\boldsymbol{p}$.

i) Accuracy. Given a placement strategy $\boldsymbol{p}$, a_{m+1} is the accuracy by using the original model exit and a_j is the accuracy by using each remaining exit j, $1 \le j \le m$. So the maximum accuracy drop caused by early exits is $\Delta_{acc}(\boldsymbol{p}) = a_{m+1} - \min(\{a_j | p_j = 1\})$. We introduce the first constraint:

$$\boldsymbol{C_{acc}}: \ \Delta_{acc}(\boldsymbol{p}) \le \alpha, \tag{3}$$

where α represents the maximum accuracy drop allowed.

ii) Pareto efficiency. Given a set of model variants, the general trend is that a larger variant has a larger delay but higher accuracy. However, we observe that it is possible for a larger variant (with larger delay) to be less accurate than another smaller variant. So such large variants can be replaced by small ones, and we aim to further rule out these anomalous variants. To this end, we can plot the accuracy-delay pairs of each exit, and their upper envelop forms a Pareto frontier of the delay and accuracy, as shown in Figure 7(b). Any points falling within (below) the Pareto frontier can be excluded, since for each of such points, we can find another point on the frontier that achieves a similar accuracy but with a shorter delay. We thus have the second constraint:

$$\boldsymbol{C_{par}}: \ \{j | p_j = 1\} \subseteq PF, \tag{4}$$

where $PF = \{j | \forall k \ne j, t_j < t_k \vee a_j \ge a_k\}$ is the Pareto frontier formed by the accuracy and execution delay. This constraint ensures that only exits along the Pareto frontier are selected, thus maintaining the relationship in which larger variants have longer delays but higher accuracy.

iii) Memory. Under placement policy $\boldsymbol{p}$ for early exits, the total size of all model variants is the sum of each chunk and preserved early exits. Therefore, we have $Mem(\boldsymbol{p}) = \sum_{i=1}^{m'} sc_i + \sum_{j \in \boldsymbol{p}} se_j$, where $m' = \max(\{j | p_j = 1\})$ is the number of chunks, and sc_i and se_j are the sizes of chunk i and exit j, respectively. Hence, to control the overall memory cost of all model variants, we define the third constraint on memory below:

$$\boldsymbol{C_{mem}}: Mem(\boldsymbol{p}) \le \beta, \tag{5}$$

where β is the maximum overall size allowed for the DNN.

iv) Objective. A preempted DNN may need to be executed again close to its deadline. Thus, the variants we generate should provide the flexibility to allow the remaining execution to be completed as quickly as possible. Considering this unique requirement due to preemption, we can measure each chunk's execution delay t'_j to its nearest exit in advance, and minimize the average t'_j across all chunks when determining $\boldsymbol{p}$, which helps minimize the deadline miss rate of DNN tasks. Therefore, we have the following objective:

$$\min_{\boldsymbol{p}} \frac{1}{m'} \sum_{j=1}^{m'} t'_j, \tag{6}$$

subjected to $\boldsymbol{C_{acc}}$, $\boldsymbol{C_{par}}$ and $\boldsymbol{C_{mem}}$ three constraints.

3.2.3 Summary of offline preprocessing. When each (pretrained) DNN task is registered in Pantheon, it is first sliced into chunks. Then, we add an early exit after to each chunk (except the last two chunks) and train them with the original DNN parameters frozen by minimizing the loss below:

$$\mathcal{L} = \{\mathcal{L}_0(\hat{y}_j, y) | \forall j \in [1, m]\}, \tag{7}$$

where $\mathcal{L}_0$ is the original loss function of the DNN. After this training, we solve Eq. (6) subjected to Eqs. (3)–(5) to obtain the placement strategy $\boldsymbol{p}$ and only keep each exit j, whose $p_j = 1$. These m^* preserved exits essentially form m^* variants for this DNN model, which are stored in the DNN variant pool to be used by the preemptive scheduling algorithm.

Note that model slicing and variant generation are both one-time effort when each DNN is registered in the system.

4 IMPLEMENTATION

Hardware. We develop Pantheon on NVIDIA Jetson Xavier NX equipped with a 384-core Volta GPU to evaluate its performance. In addition, we also deploy Pantheon on Jetson Nano of a 128-core Maxwell GPU to examine its performance on a lower-end device and a small autonomous car equipped with a Jetson Xavier NX, as shown in Figure 8, to perform a field case study of smart driving.

Software. We develop the offline module of Pantheon, responsible for slicing each DNN and adding early exits, using Python 3.8 and

[5]The last chunk is the original exit and the second last is the layer connects to the original exit. We do not need to add extra exits to them.

Application	Task Type	Dataset	DNN Model	Deadline (ms)	Number of Exits	Accuracy (%)
Smart traffic	Traffic light detection	Indoor smart traffic [40]	SSDLite [56]	20 ms	4	98.20
	Traffic sign classification	GTSRB [22]	MobileNetv2 [56]	31 ms	8	97.75
Service robot	Face Detection	FDDB [27]	SSDLite [56]	20 ms	2	87.02
	Age Classification	Adience [10]	ResNet18 [21]	28 ms	2	71.67
	Gender Classification	Adience [10]	VGG16 [57]	31 ms	4	83.74
	Emotion Recognition	FER [15]	ResNet50 [21]	31 ms	4	69.17
UAV ground station	Wildfire Detection	Wildfire Smoke [55]	SSDLite [56]	20 ms	1	91.41
	Wildlife Identification	Oregon Wildlife [48]	GoogLeNet [60]	31 ms	2	87.45
	Scene Recognition	Scene-15 [12]	ResNet34 [21]	31 ms	2	81.80

Table 1: Details of the nine real-time DNN tasks used for evaluation.

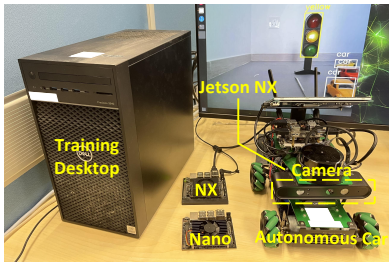

Figure 8: Devices used for the evaluation.

PyTorch. After the offline preprocessing, each DNN is converted from PyTorch format to TorchScript format and stored in the DNN pool of the online runtime. For efficient execution, we implement the online runtime and the scheduling algorithm using C++ and LibTorch, *i.e.*, the C++ front-end of PyTorch. The scheduler and worker of our runtime run in separate threads, both having access to the runtime queue protected by a Mutex. The scheduler runs the scheduling algorithm once a new task enters the queue. The worker keeps dispatching DNN chunks in the runtime queue to the underlying deep learning framework for execution. The memory manager provides APIs for adding new input and returning results to upper-layer applications, as well as guidance for context switching to the worker. The entire runtime is developed as a middleware between the deep learning framework and applications without necessitating modification to the framework and GPU driver.

Application tasks. In the experiments, nine DNN tasks (Table 1) from three applications are used to evaluate Pantheon:

1) Smart traffic: it involves 4 tasks, including 1) two traffic light detection using two SSDLites [56] and 2) two traffic sign classification using two MobileNetv2s [56] from the video frames captured by different cameras on vehicle [42].

2) Service robot: it involves 4 tasks, including 1) a face detection using SSDLite [56], 2) an age classification using ResNet34 [21], 3) a gender classification using VGG16 [57], and 4) an emotion recognition using ResNet50 [21], which enable a robot to better interact with different users.

3) UAV ground station: it involves 3 tasks, including 1) a wildfire detection using SSDLite [56], 2) an animal identification using GoogLeNet [60], and 3) a scene recognition using ResNet18 [21] for monitoring the outdoor environment [52].

When we generate DNN variants (offline) for each application, the parameter α of accuracy constraint C_{acc} is set to 1% for smart traffic and robot. For UAV ground station, we find that 1% is too tight, which makes DNN tasks unschedulable. and we set it to 2% for this application. The parameter β of memory constraint C_{mem} is set to 120% of the original model size. For each application, the total sizes of their DNNs are 20.4, 246.7 and 75.3 MB, which are increased to 23.9, 280.4 and 90.9 MB respectively after adding exits. In three applications, each task is released at 30 frames per sec and the jitter of releasing each task is within ±1 ms. The deadline for each task is set to 33 ms minus the pre-processing and post-processing time it requires [42]. Additionally, we run an AlexNet as the best-effort task in each application.

5 EVALUATION

1) Methods. We compare Pantheon with the following methods:

Rate-monotonic scheduling (RMS): it assigns higher priority to the task with a shorter execution duration, a typical scheduling algorithm in the real-time system domain [32].

Deadline-monotonic scheduling (DMS): it assigns the priority for tasks based on their closeness to deadline, another popular scheduling algorithm in real-time systems [66].

RTm-DL: the state-of-the-art DNN task scheduling on mEdge GPUs [42]. It adopts the Multi-Objective Evolutionary Algorithm to derive the global optimal scheduling strategy for priority assignment and model scaling with the assistance of a pre-trained random forest to predict the deadline miss rate for each scheduling strategy.

While continuously scaling down a model reduces model size and execution latency, it can also affect model accuracy. RTm-DL considers this accuracy constraint, and we use the same accuracy constraint as Pantheon as described above for a fair comparison in the evaluation. Furthermore, the operating frequency of GPU is fixed at the highest level for all methods throughout the evaluation.

2) Performance Metrics. Two main metrics are used to quantify the performance of each method. Before describing them, we clarify one concept used in both metrics:

- Job: when DNN tasks (Table 1) are periodically released to run, we denote the instance of each task as a job.

Deadline miss rate (DMR): this is an important metric to quantify the scheduling performance for real-time tasks with deadlines [42], which is computed as the ratio of the number of jobs (task instances) that miss deadlines and the total number of jobs executed during the evaluation.

Model accuracy: since different DNN models have different accuracies, we compute the relative accuracy of each DNN for clear illustration, which is defined as the ratio between the accuracy achieved by its variant and the accuracy achieved by the original

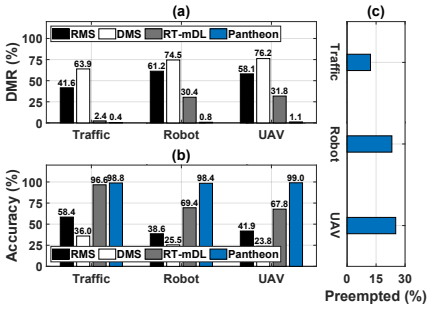

Figure 9: (a) DMR and (b) accuracy achieved by each method. (c) Ratio of the preempted jobs over the total number of jobs executed.

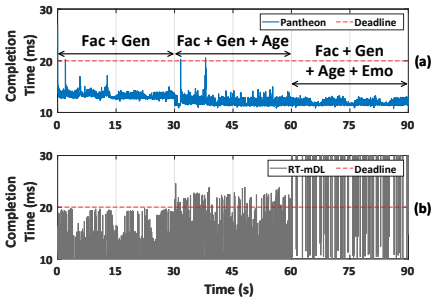

Figure 10: Completion time of "Fac" task achieved by (a) Pantheon and (b) RT-mDL under different numbers of concurrent tasks.

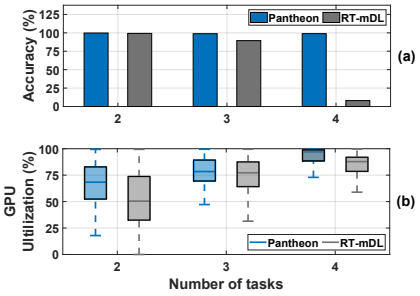

Figure 11: (a) Accuracy of "Fac" task and (b) GPU utilization achieved by Pantheon and RT-mDL under different numbers of tasks in Figure 10.

model in Table 1. In addition, the accuracy of one job is counted as zero if it misses its deadline.

5.1 Overall Performance

We first examine the overall performance of each method.

Performance. Figure 9 shows the deadline miss rate (DMR) and accuracy achieved by each method.

Deadline miss rate. From Figure 9(a), we can see that for concurrent DNN tasks, the classical real-time scheduling methods RMS and DMS become much less effective, leading to high deadline miss rates. RT-mDL can perform model scaling and execution scheduling to better accommodate multiple DNN tasks and significantly improve the deadline miss rate compared with RMS and DMS. However, due to its non-preemptive nature, RT-mDL can be affected by workload dynamics, which in turn could compromise the predefined schedule and result in missed deadlines. In contrast, Pantheon leverages preemption and can further reduce deadline miss rate to 0.39–1.10% in three applications, outperforming RMS, DMS, and RT-mDL by 92.51–98.98%, respectively.

Accuracy. As real-time task output that misses the deadline becomes invalid [66], the accuracy of a job is counted as zero if it misses the deadline. Thus, the accuracy of RMS and DMS is relatively low in Figure 9(b) due to their high deadline miss rates. RT-mDL significantly surpasses RMS and DMS in accuracy, while Pantheon can achieve even higher accuracy than RT-mDL in all three applications. Overall, Pantheon improves the accuracy by 69.1–154.8%, 174.7–315.5% and 2.2–46.0% compared to RMS, DMS, and RT-mDL, respectively.

Occurrence of preemption. To further delve into the performance gains of Pantheon, we count how often the preemption occurs in Figure 9(c). In the traffic application, the workload of DNN tasks is relatively lower than other two applications, and about 12% jobs need to be preempted, which already leads to substantial performance gains achieved by Pantheon than other methods. As the need of preemption further increases, more performance gains are obtained in Figure 9, indicating the efficacy of the Pantheon design.

Performance under different tasks. In this experiment, we further compare the system performance over time when we gradually add DNN tasks. Due to the page limitation, we mainly compare Pantheon with the state-of-the-art RT-mDL.

Deadline miss rate. Initially, only face detection ("Fac") and gender classification ("Gen") tasks in the robot application are running. Figure 10 shows the completion time of each "Fac" job and horizontal line is the deadline. When the completion time is lower than deadline, this job is finished on time. Both methods can catch up the deadline well, but the completion time of Pantheon is more stable.

When the task of age classification ("Age") is further added, RT-mDL starts to experience frequent deadline misses, while Pantheon maintains small and stable completion times. When the emotion classification ("Emo") task is finally added, Pantheon can select an appropriate model variant to ensure good compliance with the deadline. However, many jobs in RT-mDL miss their deadlines. In the last stage, the plot of RT-mDL appears sparser than that of Pantheon, because, for each job, if it already misses its deadline as soon as it starts running, we ignore it without execution (executing such job affects the execution of subsequent jobs). Many jobs of RT-mDL miss deadlines due to this reason, which are thus excluded in the figure and results in the sparse plot.

Accuracy. In Figure 11(a), we show the accuracy of "Fac" achieved by two methods under each setting in Figure 10. Pantheon performs consistent well, while the accuracy of RT-mDL drops when DMR is high in the last two settings.

GPU utilization. In Figure 11(b), we show the GPU utilization for two methods. In general, they achieve similar utilization in each setting, *e.g.*, 50.85–78.50% by Pantheon and 46.10–80.05% by RT-mDL. Figure 10–11 suggest that while consuming similar computation power, effective preemption can further ensure the timeliness of concurrent DNN tasks.

5.2 Ablation Study

Next, we conduct an ablation study to evaluate the effectiveness of two main designs proposed in Pantheon. To this end, we develop two intermediate versions of Pantheon:

- **Pantheon-w/o-ee**: it disables the capability to switch model variants once DNN execution begins. In other words, after a model is resumed after preemption, this version still uses the previous model variant for execution, which has a higher chance to miss deadline.

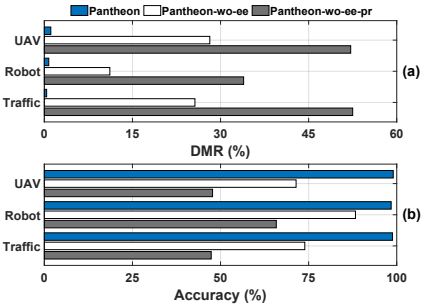

Figure 12: Ablation study for (a) DMR and (b) accuracy achieved by different versions of Pantheon in the three applications.

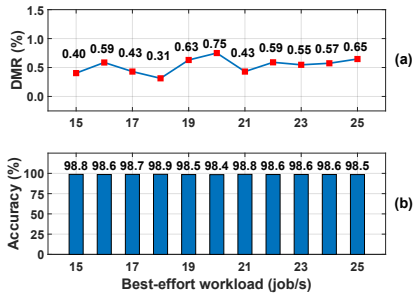

Figure 13: (a) Deadline miss rate and (b) accuracy of real-time DNN tasks achieved by Pantheon when the best-effort task workload changes.

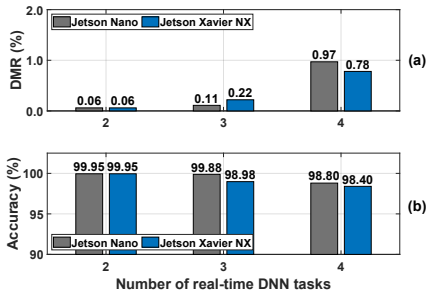

Figure 14: (a) Deadline miss rate and (b) accuracy of Pantheon under different numbers of concurrent real-time tasks on Nano and Xavier NX.

- **Pantheon-w/o-ee-pr**: it further disables the chunk-level preemption capability. The scheduling algorithm still sorts tasks based on their priorities to provide a model-wise (rather than chunk-wise) preemption.

We conduct the ablation study for all three applications. Figure 12 shows that without flexible variant adaptation, the deadline miss rate of Pantheon-w/o-ee is increased by 14–65× and the accuracy drops by 10.29–27.48%. Without chunk-level preemption, Pantheon-w/o-ee-pr further increases 44–134× deadline miss rate and degrades 33.12–52.07% accuracy. Figure 12 indicates the efficacy of our technical design for Pantheon in this paper.

5.3 Micro-benchmarks

In this subsection, we conduct micro-benchmark experiments to evaluate Pantheon under different settings.

Workload of best-effort task. To examine the impact of best-effort task, we run all four real-time DNN tasks in the robot application and change the best-effort workload of AlexNet from 15 jobs/sec to 25 jobs/sec. Since best-effort workloads are processed by low-priority GPU streams, computing resources are prioritized for real-time tasks. Thus, as shown in Figure 13(a), under different best-effort workloads, the deadline miss rates of all real-time tasks are stable, with the average and variance of 0.54% and 0.02% respectively. Figure 13(b) shows that their accuracy is also stable, with the average and variance being 98.64% and 0.02% respectively.

Different devices. In this experiment, we further deploy Pantheon on Jetson Nano, which has lower computing power than Xavier NX. To achieve similar workloads on the Nano as on Xavier NX, we scale the release rate and deadline of each task based on the difference in computing power between the two devices, following the approach in existing work [42]. Figure 14 shows the results when we vary the number of real-time DNN tasks in the robot application. For reference, we also plot the performance on Xavier NX. We can see that Pantheon achieves a deadline miss rate of 0.06–0.97% and an accuracy of 98.80–99.95% on Nano, which is similar to what is achieved on Xavier NX, which is 0.06–0.78% and 98.80–99.95% for deadline miss rate and accuracy, respectively.

Non-periodic task arrivals. In practice, the arrival of some DNN tasks may follow non-periodic patterns. For instance, the object detection task periodically works, but the number of objects detected varies. As a result, the number of jobs launched to process the detected objects is not fixed, leading to a non-periodical arrival rate. In this experiment, we aim to examine the Pantheon performance under such a setting. Another importance of this experiment is that burst job arrivals may occur under non-periodic arrival patterns.

In the three applications investigated before, each of them has the detection DNN task. In this experiment, we run the detection task periodically and change the arrival time of other tasks to be non-periodic, which follows a Poisson distribution so that the arrival rate can be quantified. Figure 15(a) plots the Poisson distributions used in this experiment with the expected values of λ from 10 to 25. Figure 15(b) shows that Pantheon achieves small deadline miss rate (0.20–0.66%) under moderate non-periodic arrival rates (*e.g.*, $\lambda = 10$ or 15). The rate increases to 1.14% on average when λ increased to 20. Under a high non-periodic arrival rate ($\lambda = 25$), where the peak number of concurrent jobs in the runtime queue is 9, Pantheon can still achieve a relatively small deadline miss rate, *e.g.*, 2.11% on average. Figure 15(c) shows the accuracy achieved under different λ values, where the accuracy is 97.27–99.61%, 96.22–99.54% and 96.22–99.65% in three applications, respectively. These results show that Pantheon performs well even when burst task arrivals occur.

5.4 Case Study

We further conduct a case study for a field evaluation of Pantheon. We build a 2m × 3m indoor smart traffic test field, as illustrated in Figure 16(a–b). The autonomous car is equipped with a camera and a Jeston Xavier NX. In the field, we set up six traffic lights distributed at the four corners and in the middle of the long side, and toy cars are randomly distributed along the road. In this experiment, our car aims to detect traffic-related objects and recognize traffic signs and scenes at the same time. Specifically, the car runs an SSDLite to detect five types of objects, including the other cars and the four statuses of the traffic lights (red, yellow, green and unlit) from the captured video frames from the camera on the car, as shown in Figure 16(c). Before the experiment, the car orbits the field to collect eight two-minute videos, which are labeled manually to train SSDLite. The video frame rate is 30. Since the car has only one camera, similar to the setting in existing work [42], we preload video frames to the car for other two tasks (traffic sign and scene recognition), and the car concurrently runs a MobileNetv2 and a ResNet34 for them respectively during testing. In the case study,

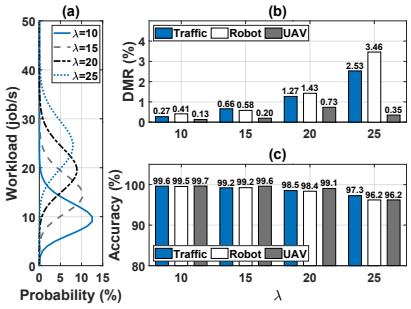

Figure 15: (a) Four Poisson distributions. (b) DMR and (b) accuracy of Pantheon with the task arrival rates following these distributions.

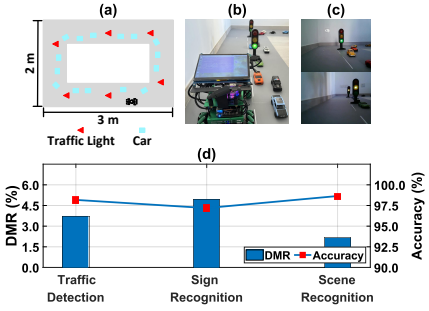

Figure 16: (a) Layout of testing field. (b) Testbed setup. (c) Examples of video frames captured. (d) DMR and accuracy achieved by Pantheon.

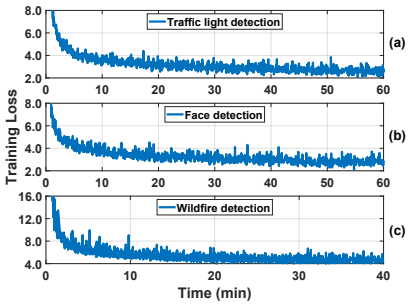

Figure 17: Training time of early exits in Pantheon on (a) traffic light detection, (b) face detection, and (c) wildfire detection three different tasks.

the deadlines of SSDLite, MobileNetv2 and ResNet34 are 18, 30 and 30 ms respectively, which are set as the release period minus the time required for pre-processing and post-processing.

Figure 16(d) shows that Pantheon achieves a good deadline miss rate of 2.15–4.91% on three tasks (3.59% on average), which is slightly higher than that in §5.1 since we set shorter deadlines to accommodate stronger delay jitter in video pre- and post-processing in the real system. Meanwhile, Pantheon also achieves good accuracy (94.4–97.2%) on three tasks.

5.5 System Overhead

Finally, we examine the overhead of Pantheon, including the training time of early exits, the latency of the scheduling algorithm and power consumption in this subsection.

Training time. In Pantheon, we add early exits to each pre-trained DNN model and then train all early exits for each model. Figure 17 shows the training loss versus the training time of early exits for three models used in the above evaluation. Since each model contains multiple early exits, we plot the average training loss in the figure. From the results we can see that the training of early exits converges within tens of minutes, such as 40–60 minutes for the three models in Figure 17. Through our experiments, we also find that this training time is in the same order of magnitude as the pre-training time of the models themselves, *e.g.*, the pre-training of these three models was completed in 25–35 minutes. Further considering that training early exits is a one-time effort for each model, such time overhead is not significant in practice.

Latency of the scheduling algorithm. Since this algorithm is frequently launched to provide the preemption logic, its execution latency should be small. Otherwise, it will affect the correctness of the task order and consequently the effectiveness of preemption. Overall, our algorithm design is efficient, which can complete the scheduling in the micro-second level. When the number of jobs to schedule is small (*e.g.*, $\leq$ 4), the scheduling takes 46.17 μs to complete on average. When the number of jobs is moderate (*e.g.*, $\leq$ 8), the latency is 130.81 μs on average. When the number of jobs is large (*e.g.*, up to 11), which is not common in practice, the latency is still small, *e.g.*, 218.86 μs on average. For clarity, we show them separately in Figure 18(a).

Power consumption. As the preemptive scheduling introduces

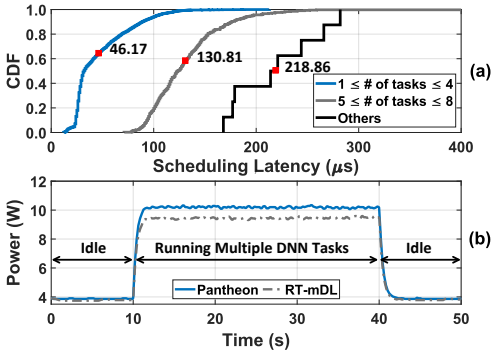

Figure 18: System overhead. (a) CDF of latency in completing the scheduling algorithm once, and (b) power consumption.

additional computations, we further investigate the device power consumption when Pantheon is used, which is measured using the INA3221 power monitor on Xavier NX. Since RT-mDL does not introduce additional online computations for scheduling, we also measure its power consumption as a baseline. In Figure 18(b), the device power consumption is about 4.01 W in the idle state. During the execution of DNN tasks, the power consumption of RT-mDL is mainly for DNN execution, which is 9.73 W on average. Due to the preemptive scheduling, the power consumption of Pantheon is increased by 4.3% only from 9.73 W to 10.15 W, which is much smaller than the energy used for the DNN execution.

6 DISCUSSION

Model profiling. In the offline preprocessing phase of Pantheon, the performance of each model variant (achieved with different early exits in Pantheon) is profiled, and the profiled results are used to place early exits. Given actual inputs and device runtime states, the performance of a model variant (*e.g.*, accuracy) may differ from the profiled one. However, profiling is still widely adopted in mobile edge systems and application designs, such as [5, 11, 19] and Pantheon in this paper, as profiling often only intends to gain a prior knowledge of the expected performance of models and provide a practical reference for runtime operations. For example, Pantheon uses it as a reference to select the appropriate model variant during

preemptive scheduling in this paper.

Additional memory access. Preemptive scheduling inevitably incurs additional memory accesses, which may slow down task inference. If we load model variants into memory from external storage via online memory allocation every time, this may introduce long delays in model inference that occurs on every context switch. Therefore, we prioritize latency by exchanging some memory cost (overseen by the memory constraint in early exit placement optimization) to avoid this delay. On the other hand, when preemption occurs, the intermediate results of the current task should be stored from the cache to the memory so that execution of the task can be resumed later. This delay cannot be completely avoided. Hence, we propose an efficient model slicing design to reduce it.

Future plans. In this paper, we mainly use CNN as an example to introduce how to apply nested redundancy to achieve efficient preemptive GPU scheduling. Specifically, unlike previous studies that explored compressed models for fixed-deadline inference [19, 28, 42], we exploit nested redundancy under a new optimization framework with early exits to enable the model to reuse the results before preemption and adjust the remaining sliced chunks to meet the deadline while maintaining good accuracy. Since early exit is mainly for CNN tasks, as an important future work of this paper, we will explore extending the Pantheon design to other types of deep learning tasks, such as transformers. In addition, we will also investigate the performance of Pantheon on more types of devices and deep learning frameworks in the future.

7 RELATED WORKS

DNN task scheduling on GPUs. Scheduling methods [4, 20, 36] in real-time systems can be applied to DNN scheduling on mEdge GPUs. However, they often fall short in efficiency as they do not consider the unique characteristics of DNN tasks and GPU architectures. Consequently, recent research on mEdge devices [35] exploit the multi-stream optimization for better parallelism between multiple DNNs [70], which still does not guarantee the timeliness of multitasking. However, multi-stream optimization is orthogonal to preemptive scheduling. It may further improve GPU throughput on top of preemption, but will significantly increase the profiling overhead (*i.e.*, covering all combinations of task parallelism and different levels of stream allocation even for the same set of tasks running in parallel) and the search space of online preemptive scheduling. RT-mDL [42] compresses DNNs based on their deadlines, but as a non-preemptive solution, workload dynamics could affect the execution delay of each task, which in turn may compromise the predefined schedule and result in missed deadlines.

For efficient task processing, mobile edge GPUs offer limited GPU stream priorities (§2). Deep learning frameworks aggregate them into high and low two priorities and follow a FIFO strategy to dispatch DNN tasks to GPU, using the priority specified by the developer. Although NVIDIA claims to support preemption from the Pascal architecture, there is no publicly available information or usable programming interface [16]. Therefore, recent work leverages the two-tier priorities to enable coarse-grained GPU-preemptive DNN inferences, allowing real-time tasks (given a high priority) to preempt best-effort tasks (given a low priority). For instance, DART [66] proposes a pipeline-based scheduling architecture, and another method [16] further improves preemption speed. However, these existing methods, which only support two-tier preemption, struggle to accommodate the trend of more concurrent real-time DNN tasks in emerging applications, which have diverse and strict latency requirements. These tasks should also be preempted among each other according to their respective deadlines.

DNN task scheduling on other platforms. Traditional mobile devices and desktop computers have not specifically provide GPU preemption. On traditional mobiles, their GPUs are only comparable to CPUs in handling DNNs [62]. Hence, recent mobile designs primarily focus on coordinating DNN processing between processors [14, 17, 29, 30] or coexisting with other tasks like video rendering [69]. In contrast, mobile edge GPUs need to handle DNN tasks independently and require a dedicated solution for multi-DNN inference. Desktops, which rarely encounter concurrent DNNs, mainly follow FIFO as well [54]. High-performance platforms, which often have multiple GPUs to handle complex concurrent DNN tasks [37], typically focus on allocating a dedicated or multiple GPUs per model [7, 37]. However, on mEdge devices, it is mainly multiple DNNs competing for one GPU.

DNN model processing. To leverage the structural and redundant characteristics of DNNs, existing studies introduce various methods to process DNNs before deployment to facilitate improved scheduling and resource allocation. One typical method is DNN slicing [18, 19, 32, 41, 66], which divides a DNN into small chunks with the aim of assigning chunks to different processors based on their affinity [32, 41, 66], reducing communication cost in offloading [18, 25], maintaining display stability [69], etc. Moreover, the redundancy of DNNs is utilized by existing work [32, 42] to adapt DNN workloads for varying deadlines. Our design is inspired by them, but we propose new designs for reducing the latency of context switching and adjusting the remaining DNN structure at runtime to meet shortened deadlines due to preemption.

8 CONCLUSION

This paper introduces Pantheon, a new preemption design for multi-DNN inference in mobile edge GPUs. The design of Pantheon reveals that the two-tier GPU stream prioritization available on mobile edge devices is adequate to enable such services. These two tiers of priorities are primarily used to distinguish between real-time and best-effort tasks. Preemption between real-time tasks, whose priorities change over time, can be further achieved through software design. This includes an online runtime with comprehensive preemption logic and innovative scheduling, and offline DNN processing. Pantheon does not need to modify deep learning frameworks and GPU drivers, making it easy to deploy. Extensive experiments show significant improvements in system performance compared to state-of-the-art methods.

ACKNOWLEDGEMENT

We sincerely thank anonymous reviewers for their helpful review comments to improve the quality of this paper. This work is supported by the GRF grant from Research Grants Council of Hong Kong (CityU 11202623) and the APRC grant from City University of Hong Kong (Project No. 9610633). Zimu Zhou and Zhenjiang Li are the corresponding authors.

REFERENCES

[1] Tensorflow. https://www.tensorflow.org/, 2023.

[2] Tanya Amert, Nathan Otterness, Ming Yang, James H Anderson, and F Donelson Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *Proc. of IEEE RTSS*, 2017.

[3] ARM. Arm® cortex®-a series programmer's guide for armv8-a. https://developer.arm.com/documentation/den0024/a, 2015.

[4] Neil C Audsley, Alan Burns, and Andy J Wellings. Deadline monotonic scheduling theory and application. *Control Engineering Practice*, 1993.

[5] Soroush Bateni and Cong Liu. Neuos: A latency-predictable multi-dimensional optimization framework for dnn-driven autonomous systems. In *Proc. of USENIX ATC*, 2020.

[6] Jiani Cao, Chengdong Lin, Yang Liu, and Zhenjiang Li. Gaze tracking on any surface with your phone. In *Proc. of ACM SenSys*, 2022.

[7] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *Proc. of USENIX USDI*, 2017.

[8] Xianzhong Ding, Wan Du, and Alberto Cerpa. Octopus: Deep reinforcement learning for holistic smart building control. In *Proc. of ACM BuildSys*, 2019.

[9] Jonatan S Dyrstad and John Reidar Mathiassen. Grasping virtual fish: A step towards robotic deep learning from demonstration in virtual reality. In *Proc. of IEEE ROBIO*, 2017.

[10] Eran Eidinger, Roee Enbar, and Tal Hassner. Age and gender estimation of unfiltered faces. *IEEE Transactions on information forensics and security*, 2014.

[11] Biyi Fang, Xiao Zeng, and Mi Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proc. of ACM MobiCom*, 2018.

[12] Li Fei-Fei and Pietro Perona. A bayesian hierarchical model for learning natural scene categories. In *Proc. of IEEE CVPR*, 2005.

[13] Ernestine Fu, David Hyde, Srinath Sibi, Mishel Johns, Martin Fischer, and David Sirkin. Assessing the effects of failure alerts on transitions of control from autonomous driving systems. In *Proc. of IEEE IV*, 2020.

[14] Petko Georgiev, Nicholas D Lane, Kiran K Rachuri, and Cecilia Mascolo. Leo: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *Proc. of ACM MobiCom*, 2016.

[15] Ian J Goodfellow, Dumitru Erhan, Pierre Luc Carrier, Aaron Courville, Mehdi Mirza, Ben Hamner, Will Cukierski, Yichuan Tang, David Thaler, Dong-Hyun Lee, et al. Challenges in representation learning: A report on three machine learning contests. In *Proc. of ICONIP*, 2013.

[16] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *Proc. of USENIX OSDI*, 2022.

[17] Myeonggyun Han and Woongki Baek. Herti: A reinforcement learning-augmented system for efficient real-time inference on heterogeneous embedded systems. In *Proc. of IEEE PACT*, 2021.

[18] Myeonggyun Han, Jihoon Hyun, Seongbeom Park, Jinsu Park, and Woongki Baek. Mosaic: Heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference. In *Proc. of IEEE PACT*, 2019.

[19] Rui Han, Qinglong Zhang, Chi Harold Liu, Guoren Wang, Jian Tang, and Lydia Y Chen. Legodnn: block-grained scaling of deep neural networks for mobile vision. In *Proc. of ACM MobiCom*, 2021.

[20] Yifan Hao. *Deep intelligence as a service: A real-time scheduling perspective*. PhD thesis, University of Illinois at Urbana-Champaign, 2019.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. of IEEE CVPR*, 2016.

[22] Sebastian Houben, Johannes Stallkamp, Jan Salmen, Marc Schlipsing, and Christian Igel. Detection of traffic signs in real-world images: The German Traffic Sign Detection Benchmark. In *Proc. of IJCNN*, 2013.

[23] Chuang Hu, Wei Bao, Dan Wang, and Fengming Liu. Dynamic adaptive dnn surgery for inference acceleration on the edge. In *Proc. of IEEE INFOCOM*, 2019.

[24] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens Van Der Maaten, and Kilian Q Weinberger. Multi-scale dense networks for resource efficient image classification. In *Proc. of ICLR*, 2018.

[25] Kai Huang and Wei Gao. Real-time neural network inference on extremely weak devices: agile offloading with explainable ai. In *Proc. of ACM MobiCom*, 2022.

[26] Modor Intelligence. Gpu market size & share analysis – growth trends & forecasts (2023 - 2028). https://www.mordorintelligence.com/industry-reports/graphics-processing-unit-market, 2023.

[27] Vidit Jain and Erik Learned-Miller. Fddb: A benchmark for face detection in unconstrained settings. Technical Report UM-CS-2010-009, University of Massachusetts, Amherst, 2010.

[28] Seunghyeok Jeon, Yonghun Choi, Yeonwoo Cho, and Hojung Cha. Harvnet: Resource-optimized operation of multi-exit deep neural networks on energy harvesting devices. In *Proc. of ACM MobiSys*, 2023.

[29] Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. Band: coordinated multi-dnn inference on heterogeneous mobile processors. In *Proc. of ACM MobiSys*, 2022.

[30] Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. Codl: efficient cpu-gpu co-execution for deep learning inference on mobile devices. In *Proc. of ACM MobiSys*, 2022.

[31] Shiqi Jiang, Zhiqi Lin, Yuanchun Li, Yuanchao Shu, and Yunxin Liu. Flexible high-resolution object detection on edge devices with tunable latency. In *Proc. of ACM MobiCom*, 2021.

[32] Woosung Kang, Kilho Lee, Jinkyu Lee, Insik Shin, and Hoon Sung Chwa. Lalarand: Flexible layer-by-layer cpu/gpu scheduling for real-time dnn tasks. In *Proc. of IEEE RTSS*, 2021.

[33] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *Proc. of ACM/IEEE IPSN*, 2016.

[34] Stefanos Laskaridis, Stylianos I Venieris, Hyeji Kim, and Nicholas D Lane. Hapi: Hardware-aware progressive inference. In *Proc. of IEEE/ACM ICCAD*, 2020.

[35] Jingyu Lee, Yunxin Liu, and Youngki Lee. Parallelfusion: towards maximum utilization of mobile gpu for dnn inference. In *Proc. of ACM EMDL*, 2021.

[36] John Lehoczky, Lui Sha, and Yuqin Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proc. of IEEE RTSS*, 1989.

[37] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. In *Proc. of USENIX OSDI*, 2023.

[38] Chengdong Lin, Kun Wang, Zhenjiang Li, and Yu Pu. A workload-aware dvfs robust to concurrent tasks for mobile devices. In *Proc. of ACM MobiCom*, 2023.

[39] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. The architectural implications of autonomous driving: Constraints and acceleration. In *Proc. of ACM ASPLOS*, 2018.

[40] Neiwen Ling, Yuze He, Nan Guan, Heming Fu, and Guoliang Xing. An indoor smart traffic dataset and data collection system: Dataset. In *Proc. of ACM SenSys*, 2022.

[41] Neiwen Ling, Xuan Huang, Zhihe Zhao, Nan Guan, Zhenyu Yan, and Guoliang Xing. Blastnet: Exploiting duo-blocks for cross-processor real-time dnn inference. In *Proc. of ACM SenSys*, 2022.

[42] Neiwen Ling, Kai Wang, Yuze He, Guoliang Xing, and Daqi Xie. Rt-mdl: Supporting real-time mixed deep learning tasks on edge platforms. In *Proc. of ACM SenSys*, 2021.

[43] Jane WS Liu et al. *Real-time systems*. Pearson Education India, 2006.

[44] Luyang Liu, Hongyu Li, and Marco Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *Proc. of ACM MobiCom*, 2019.

[45] Sicong Liu, Bin Guo, Ke Ma, Zhiwen Yu, and Junzhao Du. Adaspring: Context-adaptive and runtime-evolutionary deep model compression for mobile applications. *Proceeding of the ACM on IMWUT*, 2021.

[46] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *Proc. of ACM MobiSys*, 2018.

[47] Szymon Migacz. Performance tuning guide – use onednn graph with torchscript for inference. https://pytorch.org/tutorials/recipes/recipes/tuning_-guide.html#use-onednn-graph-with-torchscript-for-inference, 2023.

[48] David Molina. Oregon wildlife. https://www.kaggle.com/datasets/virtualdvid/oregon-wildlife, 2018.

[49] NVIDIA. Cuda runtime api. https://docs.nvidia.com/cuda/cuda-runtime-api/index.html, 2023.

[50] NVIDIA. Nvidia jetson. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/, 2023.

[51] NVIDIA. Tuning cuda applications for volta. https://docs.nvidia.com/cuda/volta-tuning-guide/index.html, 2023.

[52] Lucas Prado Osco, José Marcato Junior, Ana Paula Marques Ramos, Lúcio André de Castro Jorge, Sarah Narges Fatholahi, Jonathan de Andrade Silva, Edson Takashi Matsubara, Hemerson Pistori, Wesley Nunes Gonçalves, and Jonathan Li. A review on deep learning in uav remote sensing. *International Journal of Applied Earth Observation and Geoinformation*, 2021.

[53] Xiaomin Ouyang, Xian Shuai, Jiayu Zhou, Ivy Wang Shi, Zhiyuan Xie, Guoliang Xing, and Jianwei Huang. Cosmo: contrastive fusion learning with small data for multimodal human activity recognition. In *Proc. of ACM MobiCom*, 2022.

[54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 2019.

[55] roboflow. Wildfire smoke dataset. https://public.roboflow.com/object-detection/wildfire-smoke, 2020.

[56] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proc. of IEEE CVPR*, 2018.

[57] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for

large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
[58] William Stallings. *Operating systems: internals and design principles*. Prentice Hall Press, 2011.
[59] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 2017.
[60] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proc. of IEEE CVPR*, 2015.
[61] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *Proc. of IEEE ICPR*, 2016.
[62] Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu. Asymo: scalable and efficient deep-learning inference on asymmetric mobile cpus. In *Proc. of ACM MobiCom*, 2021.
[63] Yanwen Wang, Jiaxing Shen, and Yuanqing Zheng. Push the limit of acoustic gesture recognition. In *Proc. of IEEE INFOCOM*, 2020.
[64] Zeyu Wang, Xiaoxi He, Zimu Zhou, Xu Wang, Qiang Ma, Xin Miao, Zhuo Liu, Lothar Thiele, and Zheng Yang. Stitching weight-shared deep neural networks for efficient multitask inference on gpu. In *Proc. of IEEE SECON*, 2022.
[65] Wikipedia. Preemption (computing). https://en.wikipedia.org/wiki/Preemption_(computing), 2023.
[66] Yecheng Xiang and Hyoseung Kim. Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference. In *Proc. of IEEE RTSS*, 2019.
[67] Huatao Xu, Pengfei Zhou, Rui Tan, Mo Li, and Guobin Shen. Limu-bert: Unleashing the potential of unlabeled data for imu sensing applications. In *Proc. of ACM SenSys*, 2021.
[68] Juheon Yi, Sunghyun Choi, and Youngki Lee. Eagleeye: Wearable camera-based person identification in crowded urban spaces. In *Proc. of ACM MobiCom*, 2020.
[69] Juheon Yi and Youngki Lee. Heimdall: mobile gpu coordination platform for augmented reality applications. In *Proc. of ACM MobiCom*, 2020.
[70] Zhihe Zhao, Neiwen Ling, Nan Guan, and Guoliang Xing. Miriam: Exploiting elastic kernels for real-time multi-dnn inference on edge gpu. In *Proc. of ACM SenSys*, 2023.

A APPENDIX

In the appendix, we provide the necessary guidelines for artifact evaluators to install and run Pantheon. The source codes of Pantheon are available at https://github.com/PantheonInfer/Pantheon.

A.1 Prerequisites

In this subsection, we detail the hardware and software requirements for our offline and online modules, respectively. The previous is in charge of DNN slicing and variant generation, while the latter includes the runtime and profiling parts. Also, we describe the Jetson board device settings that should be done before evaluation.

Offline modules. Offline modules operate on a desktop (with Windows, but others might also work) connected to the Jetson board with an Ethernet cable (for profiling). The main software reliance is listed as follows:

- *Python 3.8*
- *PyTorch 2.0.1*
- *PyTorch Lightning 1.1.0*
- *pymoo 0.6*

For the full list of Python packages required, you can find them on `requirements.txt` from the released source codes, and install them via `pip install -r requirements.txt`.

Online modules. To test the online modules, an NVIDIA Jetson Xavier NX Developer Kit with Jetpack 5.1.1 is required. The software requirements and installation steps are described below:

CMake. It is usually installed by default, and you should make sure that its version is equal to or newer than 3.17.5.

LibTorch. It will be available once PyTorch is installed successfully on Jetson boards. For installing PyTorch on Jetson, please refer to the official guidelines: https://docs.nvidia.com/deeplearning/frameworks/install-pytorch-jetson-platform/index.html.

OpenCV. It is usually installed by default and can be checked by

```
cd /usr/bin
./opencv_version
```

Protobuf. It should be installed via the following command:

```
sudo apt-get install protobuf-compiler
↪  libprotobuf-dev
```

spdlog. It should be installed via the following commands:

```
git clone https://github.com/gabime/spdlog.git
cd spdlog && mkdir build && cd build
cmake .. && make -j
make install
```

Device settings. The Jetson Xavier NX board is powered by a 19V 2.37A 45W AC adapter and connected to the desktop using an Ethernet cable. The NVP mode of the Jetson board should be set to `MODE_20W_6CORE` by:

```
sudo nvpmodel -m 8
```

The dynamic voltage frequency scaling governor should be disabled, and the processors' clocks should be locked to their maximums using:

```
sudo jetson_clocks
```

A.2 Usages

We provide the pre-trained DNN models on xxxxxxxxxxxxxxxxxx. The datasets that are required for the variant generation can be downloaded from xxxxxxxxxxxxxxxxxx. With the pre-trained models and dataset downloaded, Pantheon can be run following the steps:

1) DNN slicing and variant generation. In this step, we slice a given pre-trained DNN into chunks, insert early exits in between all chunks, and then train all inserted early exits. These can be done by simply running the script `offline/construct.py` with the arguments listed in Table 2.

Argument	Help
–task	Specify the DNN task.
–data_dir	The root directory of all datasets.
–log	The directory to store train loggings.
–learning_rate	Learning rate for training early exits.
–batch_size	Batch size for training early exits.
–weight_decay	Weight decay for learning rate.
–max_epochs	Number of epochs for training early exits.
–num_workers	Number of threads for loading datasets.
–gpu_id	Specify the GPU for training in case multiples are available.
–pretrain	The path to the pre-trained model weights.

Table 2: Arguments of `offline/construct.py`.

For example, to slice and generate variants for the traffic sign classification task, we can run the following script:

```
python offline/construct.py --task
↪  sign_recognition --data_dir /path/to/datasets
↪  --log /path/to/logs --max_epochs 100
↪  --weight_decay 1e-3 --pretrain
↪  /path/to/weights
```

For the argument values for each task, please refer to Table 3.

Task name	–task	–learning_rate	–batch_size	–weight_decay	–max_epochs
Traffic light detection	object_detection	1e-3	24	5e-4	50
Traffic sign classification	sign_recognition	1e-2	256	1e-3	100
Face Detection	face_detection	1e-3	24	5e-4	100
Age Classification	age_classification	1e-3	128	1e-4	200
Gender Classification	gender_classification	1e-3	128	1e-4	50
Emotion Recognition	emotion_classification	1e-2	64	5e-4	100
Wildfire Detection	wildfire_detection	1e-3	24	5e-4	100
Wildlife Identification	wildlife_recognition	1e-2	64	5e-3	100
Scene Recognition	scene_recognition	1e-2	256	1e-2	100

Table 3: `offline/construct.py` argument values for tasks.

As the training process takes a long time, we also provide the trained variants for all tasks on xxxxxxxxxxxxxxxx for downloading.

2) Model format transformation. The generated model variants are now saved in PyTorch style, which is not available to LibTorch (C++ front-end). Hence, the next step is to export the generated model

variants to JIT format by running the script `offline/export.py` with the arguments listed in Table 4.

Argument	Help
–task	Specify the DNN task.
–save	The directory to save the exported model weights.
–weights	The path to the generated variants.

Table 4: Arguments of `offline/construct.py`.

We also provide all exported variant weights in JIT format on xxxxxxxxxxxxxxxxxxxx.

3) Profiling. In this step, we profile the accuracy, latency, and memory costs of model variants. To profile the latency on the Jetson board, we should first compile the profiler on it as follows:

```
cd online/apps/profiler
mkdir build
cd build
cmake -DCMAKE_PREFIX_PATH=`python3 -c 'import
↪  torch;print(torch.utils.cmake_prefix_path)'`
↪  ..
cmake --build . --config Release
```

Then, we need to run the profiler on the Jetson board by:

```
./profiler
```

At the same time, we should run the script `offline/_profile.py` on the desktop with arguments listed in Table 5. This script with communicate with the profiler to evaluate the latency on board, evaluate the memory costs and accuracy on the desktop, and generate the profiling results as a CSV file automatically.

Argument	Help
–task	Specify the DNN task.
–data_dir	The root directory of all datasets.
–weights	The directory holding exported variants.
–host	IP for TCP/IP connection with the device (default value will do).
–port	Port for TCP/IP connection with the device (default value will do).
–buffer	Buffer size for TCP/IP connection with the device (default value will do).
–batch_size	Batch size for evaluating accuracy.
–num_workers	Number of threads for loading datasets.

Table 5: Arguments of `offline/construct.py`.

For example, we can simply run it like this:

```
python offline/_profile.py --task sign_recognition
↪  --weights path/to/weights
```

4) Early exit placement optimization. With the profiled results, in this step, we optimize the placement of early exits and remove redundant exits for actual deployment. This is done for all tasks in the same application at once by running the script `offline/plan.py`:

```
python offline/plan.py --base
↪  directory/to/variants/weights/of/all/tasks
↪  --setting path/to/application/settings --save
↪  directory/to/save/variants/for/deployment
```

The `–base` argument should be the directory holding the variants' weights in JIT format for all tasks in the specific application. The `–setting` specifies the constraint for that application, including the allowed maximum accuracy drop for each task and the memory constraints, which are given as a JSON file. We provide the settings for all applications in `experiments/settings/deploy`. The `–save` argument specifies the directory to hold the model variants for deployment, which should be uploaded to the Jetson board.

5) Deployment. The last step is to deploy the application to the Jetson board. First, we should compile the runtime on board:

```
cd online/apps/runtime
mkdir build
cd build
cmake -DCMAKE_PREFIX_PATH=`python3 -c 'import
↪  torch;print(torch.utils.cmake_prefix_path)'`
↪  ..
cmake --build . --config Release
```

Then, we start Pantheon runtime to execute the application with specified workloads

```
./runtime /path/to/models /path/to/workload
↪  /path/to/log/
```

The first argument is the path to the directory storing the variants generated in the previous step. The second argument specifies the workload, which is given in Protobuf format. We provide the workloads for each application in `experiments/settings/workload`. The last argument specifies the path to store the runtime logging, which is then used to evaluate runtime performance.

At the end, we can download the runtime logging to the desktop and run `experiments/logs/scheduling/viz.py` to evaluate the runtime performance:

```
python experiments/logs/scheduling/viz.py
↪  /path/to/log /path/to/workload
```